The Active Element Machine

Michael Stephen Fiske

Abstract A new computing machine, called an active element machine (AEM), and the AEM programming language are presented. This computing model is motivated by the positive aspects of dendritic integration, inspired by biology, and traditional programming languages based on the register machine. Distinct from the traditional register machine, the fundamental computing elements – active elements – compute simultaneously. Distinct from traditional programming languages, all active element commands have an explicit reference to time. These attributes make the AEM an inherently parallel machine, enable the AEM to change its architecture (program) as it is executing its program. Using a random bit source from the environment and the Meta command, an AEM is created that represents an arbitrary real number in $[0,1]$. Exploiting the randomness from the environment, this example is extended to an AEM that can recognize an arbitrary binary language $L \subseteq \{0,1\}^*$. Finally, an AEM finds the Ramsey number $r(3,3)$, illustrating how parallel AEM algorithms and time in the commands help compute an NP-hard problem.

1 Introduction

A new computing machine called an active element machine (AEM) and the active element machine programming language are introduced. This computing model is motivated by the positive aspects of dendritic integration, inspired by biology, and traditional programming languages based on the register machine. Distinct from the traditional register machine, the fundamental computing elements – active elements – compute simultaneously. Distinct from traditional programming languages, all active element commands have an explicit reference to time. These attributes make the AEM an inherently parallel machine and enable the AEM to change its architecture (program) as it is executing its program.

Michael Stephen Fiske
Aemea Institute, San Francisco, CA, 94129. e-mail: mf@aemea.org

1.1 Wilfrid Rall's Models of Dendritic Integration

Wilfrid Rall's research [35] in neurophysiology influenced the development of the active element machine – in particular, his work on dendritic integration and how this contributes to computation. Rall's mathematical models are thorough and complicated; Rall modelled the non-linearities of the neuron and much of his work focussed on the dendrites.

Our goal was to capture the critical computational properties of dendritic integration that use its computational power while keeping the mathematics as simple as possible. Another goal was to assure that the mathematics and computing mechanism were simple enough to implement in silicon and other kinds of hardware ([18], [19]).

Our third goal was to make the machine and language simple enough to design autonomous systems (implicitly program) with evolutionary methods or for a person to explicitly program or both. Early and current neural network models [25], [31] are complicated to program or do not have a simple programming language for designing the network. For the above reasons, implicit and explicit programmability were important criteria that influenced the design of the active element machine.

1.2 Register Machine Computation

Another part of this development comes from the formal model of the Turing machine [45] and the subsequent von Neumann architecture. (This section contains some rhetorical content as a means to motivate new notions.) Today's computers do not conceptually work much differently than these early models. Perhaps, the biggest difference is that today's computers are much faster. In the current notion of an *algorithm*, the relevant concepts of the Turing computing model (see [45] and definitions 13, 14, 15, 16) are:

- There are finite number of alphabet symbols $A = \{a_1, \dots, a_n\}$ read and written to a tape.
- There are a finite number of machine states $Q = \{q_1, \dots, q_m\}$.
- The Turing program, η, is a finite set of rules that stays fixed i.e. the rules do not change as the program executes.
- The execution of one rule represents a computational step. During this computational step, one of the rules is selected, based on the current alphabet symbol pointed to by the tape head and the current machine state. The output of the rule specifies that a new alphabet symbol or the same symbol is written to the tape, the machine moves to a new state or stays in the current state and that the tape head moves one square to the left or right.
- *Computational steps are executed sequentially with no explicit reference to time.*

In light of the above, it seems natural for the Turing machine to lead to the register machine (see [1], [33], [44]). In the register machine, a program is a finite number of instructions that are executed in a linear sequence. Further, the contents of a register is changed in one computational step, which is analogous to writing a new symbol on the tape during one computational step of the Turing machine. In the register machine, there is also no explicit reference to an absolute or relative time. Furthermore, usually one register machine instruction is executed at a time, which creates a computational bottleneck.

1.3 Explicit Representation of Time

The register machine is a programmable machine but the program is fixed during program execution. There is also no notion of explicit time in the register machine model, only the order in which instructions are executed. Rall's research does not address programmability and has no notion of commands. His models used time, dendritic integration and adaptability of the synapses. The active element machine explicitly represents time in the machine commands which enables the following useful properties.

- Parallel algorithms can be implemented in a natural way, since each active element performs computation and all of them simultaneously compute.
- Explicit time in the active element commands enhances control over the active element machine computation because the synchronization of computation among different groups of active elements can be coordinated. This coordination helps avoid race conditions that can occur in the standard programming languages that implement concurrent processes.
- The machine can change its own architecture (program) with the Meta command while it is executing.
- The Meta command enables the active element machine's complexity to increase over time.

In [30], Edward Lee proposes using explicit time in a computing model and computing applications.

> This paper argues that to realize its full potential, the core abstractions of computing need to be rethought to incorporate essential properties of the physical systems, most particularly the passage of time. It makes a case that the solution cannot be simply overlaid on existing abstractions, ... The emphasis needs to be on repeatable behavior rather than on performance optimization.

1.4 Summary

Overall, the active element machine and a programming language can be used to explicitly or implicitly program the machine. Any register machine can be computed by an active element machine. Using randomness in the environment, time and the Meta command, a finite active element machine program can represent an arbitrary real number in $[0,1]$. Building upon this example, this AEM is extended so that it can recognize an arbitrary language $L \subseteq \{0,1\}^*$. Finally, an example of an active element machine is demonstrated that finds the Ramsey number $r(3,3)$ ([22], [36]), illustrating how parallel AEM algorithms and time in the commands compute an NP-hard problem ([10], [12], [21]).

2 Machine Architecture

An active element machine is composed of computational primitives called active elements. There are three kinds of active elements: Input, Computational and Output active elements. Input active elements process information received from the environment or another active element machine. Computational active elements receive messages from the input active elements and other computational active elements firing activity and transmit new messages to computational and output active elements. The output active elements receive messages from the input and computational active elements firing activity. The firing activity of the output active elements represents the output of the active element machine. Every active element is an active element in the sense that each one can receive and transmit messages simultaneously.

Each active element receives messages, formally called pulses, from other active elements and itself and transmits messages to other active elements and itself. If the messages received by active element, E_i, at the same time sum to a value greater than the threshold, then active element E_i fires. When an active element E_i fires, it sends messages to other active elements.

Let $\mathbb{Z}$ denote the integers. Define the extended integers as $\overline{\mathbb{Z}} = \{m + kdT : m,k \in \mathbb{Z}$ and dT is a fixed infinitesimal$\}$. For more on infinitesimals, see [38].

Definition 1. *Machine Architecture*

Γ, Ω, and Δ are index sets that index the input, computational, and output active elements, respectively. Depending on the machine architecture, the intersections $\Gamma \cap \Omega$ and $\Omega \cap \Delta$ can be empty or non-empty. A machine architecture, denoted as $\mathscr{M}(\mathscr{I},\mathscr{E},\mathscr{O})$, consists of a collection of input active elements, denoted as $\mathscr{I} = \{E_i : i \in \Gamma\}$; a collection of computational active elements $\mathscr{E} = \{E_i : i \in \Omega\}$; and a collection of output active elements $\mathscr{O} = \{E_i : i \in \Delta\}$. Each computational and output active element, E_i, has the following components and properties:

- A threshold θ_i

- A refractory period r_i where $r_i > 0$.
- A collection of pulse amplitudes $\{A_{ki} : k \in \Gamma \cup \Omega\}$.
- A collection of transmission times $\{\tau_{ki} : k \in \Gamma \cup \Omega\}$, where $\tau_{ki} > 0$ for all $k \in \Gamma \cup \Omega$.
- A function of time, $\Psi_i(t)$, representing the time active element E_i last fired. $\Psi_i(t) = \sup\{s : s < t \text{ and } g_i(s) = 1\}$, where $g_i(s)$ is the output function of active element E_i and is defined below. The sup is the least upper bound.
- A binary output function, $g_i(t)$, representing whether active element E_i fires at time t. The value of $g_i(t) = 1$ if $\sum A_{ki}(t) > \theta_i$ where the sum ranges over all $k \in \Gamma \cup \Omega$ and $t \geq \Psi_i(t) + r_i$. In all other cases, $g_i(t) = 0$. For example, $g_i(t) = 0$, if $t < \Psi_i(t) + r_i$.
- A set of firing times of active element E_k within active element E_i's integrating window, $W_{ki}(t) = \{s : \text{active element } E_k \text{ fired at time } s \text{ and } 0 \leq t - s - \tau_{ki} < \omega_{ki}\}$. Let $|W_{ki}(t)|$ denote the number of elements in the set $W_{ki}(t)$. If $W_{ki}(t) = \emptyset$, then $|W_{ki}(t)| = 0$.
- A collection of input functions, $\{\phi_{ki} : k \in \Gamma \cup \Omega\}$, each a function of time, and each representing pulses coming from computational active elements, and input active elements. The value of the input function is computed as $\phi_{ki}(t) = |W_{ki}(t)| A_{ki}(t)$.
- The refractory periods, transmission times and pulse widths are positive integers; and pulse amplitudes and thresholds are integers. The time t – that these parameters are a function of i.e. $\theta_i(t), r_i(t), A_{ki}(t), \omega_{ki}(t), \tau_{ki}(t)$ – is an element of the extended integers $\overline{\mathbb{Z}}$.

Input active elements that are not computational active elements have the same characteristics as computational active elements, except they have no inputs ϕ_{ki} coming from active elements in this machine. In other words, they don't receive pulses from active elements in this machine. Input active elements are assumed to be externally firable. An external source such as the environment or an output active element from another distinct machine $\mathscr{M}(\mathscr{I}', \mathscr{E}', \mathscr{O}')$ can cause an input active element to fire. The input active element can fire at any time as long as the current time minus the time the input active element last fired is greater than or equal to the input active element's refractory period.

An active element, E_i, can be an input active element and a computational active element. Similarly, an active element can be an output active element and a computational active element. Alternatively, when an output active element, E_i, is not a computational active element, where $i \in \Delta - \Omega$, then E_i does not send pulses to active elements in this machine.

Example 1. Overlapping Pulses with Different Firing Times

Consider the four element machine where X, Y, and Z are input active elements and B is a computational active element. The parameters of the elements and their connections are shown in tables 1 and 2.

Table 1 Element Parameter Values

Element	Threshold	Refractory	Firing Times
X		1	4
Y		1	1
Z		1	2
B	10	2	5

No thresholds are shown for X, Y, and Z since they are input elements.

Table 2 Connection Parameter Values

Connection	From	To	Amplitude	Width	Transmission Time
XB	X	B	3	4	1
YB	Y	B	4	2	3
ZB	Z	B	4	2	2

Input elements X, Y, and Z are externally fired at times 4, 1 and 2, respectively. At time $t = 3$, pulses created by Y and Z are travelling to B but have not yet arrived. B does not fire. At time $t = 4$, pulses created by Y and Z arrive at B. The input to B is $A_{YB}(4) + A_{ZB}(4) = 8$ which does not exceed B's threshold. B does not fire. At time $t = 5$, pulses created by Y and Z are still at B because their pulse widths are 2. Also the pulse from X arrives. The input to B is $A_{XB}(5) + A_{YB}(5) + A_{ZB}(5) = 11$ which exceeds B's threshold. B fires at time $t = 5$. At time $t = 7$, the refractory period of B has expired. The pulses created by Y and Z have passed through B. The pulse from X is still at B. The input to B is $A_{XB}(7) = 3$ which does not exceed B's threshold. B does not fire at time $t = 7$.

The machine architecture is summarized. If $g_i(s) = 1$, this means active element E_i *fired* at time s. The *refractory period*, r_i, is the amount of time that must elapse after active element E_i just fired before E_i can fire again. The *transmission time*, τ_{ki}, is the amount of time it takes for active element E_i to find out that active element E_k has fired. The *pulse amplitude*, A_{ki}, represents the strength of the pulse that active element E_k transmits to active element E_i after active element E_k has fired. After this pulse reaches E_i, the *pulse width* ω_{ki} represents how long the pulse lasts as input to active element E_i. At time s, the *connection* from E_k to E_i represents the triplet $(A_{ki}(s), \omega_{ki}(s), \tau_{ki}(s))$. If $A_{ki} = 0$, then there is no connection from active element E_k to active element E_i.

3 Active Element Machine Programming Language

In this section, a programming language is defined to explicitly program an active element machine and to change the machine architecture as program execution proceeds. It is helpful to define a programming language, influenced by S-expressions.

There are five types of commands: `Element`, `Connection`, `Fire`, `Program` and `Meta`.

Definition 2. *AEM Program*
In Backus-Naur form, an AEM program is defined as follows.

```
<AEM_program> ::= <cmd_sequence>

<cmd_sequence> ::= "" | <AEM_cmd><cmd_sequence>
                 | <program_def><cmd_sequence>

<AEM_cmd> ::= <element_cmd> | <fire_cmd> | <meta_cmd>
            | <cnct_cmd> | <program_cmd>
```

Definition 3. *AEM Symbols and Extended Integer Expressions*
In Backus-Naur form, the AEM symbols are defined as follows.

```
<ename> ::= "" | <int>| <symbol>

<symbol> ::= <symbol_string> | (<ename> . . . <ename>)

<symbol_string> ::= "" | <char_symbol><str_tail>

<str_tail> ::= "" | <char_symbol><str_tail> | 0<str_tail>
             | <pos_int><str_tail>

<char_symbol> ::= <letter> | <special_char>

<letter> ::= <lower_case> | <upper_case>

<lower_case> ::= a|b|c|d|e|f|g|h|i|j|k|l|m
               |n|o|p|q|r|s|t|u|v|w|x|y|z

<upper_case> ::= A|B|C|D|E|F|G|H|I|J|K|L|M
               |N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<special_char> ::= "" | _
```

These rules represent the extended integers, addition and subtraction.

```
<int> ::= <pos_int> | <neg_int> | 0

<neg_int> ::= −<pos_int>

<pos_int> ::=  <non_zero><digits>

<digits> ::= <numeral> | <numeral><digits>

<non_zero> ::= 1|2|3|4|5|6|7|8|9

<numeral> ::= "" | <non_zero> | 0

<aint> ::= <aint> <math_op> <d> | <d> <math_op> <aint> | <d>

<math_op> ::= + | -

<d> ::= <int> | <symbol_string> | <infinitesimal>

<infinitesimal> ::= dT
```

Definition 4. `Element`

An `Element` command specifies the time when an active element's values are updated or created. This command has the following Backus-Naur syntax.

```
<element_cmd> ::= (Element (Time <aint>) (Name <ename>)
               (Threshold <int>)(Refractory <pos_int>)(Last <int>))
```

The keyword `Time` tags the time integer value s at which the element is created or updated. If the name symbol value is `E`, the keyword `Name` tags the name `E` of the active element. The keyword `Threshold` tags the threshold $\theta_E(s)$ assigned to `E`. `Refractory` tags the refractory value $r_E(s)$. The keyword `Last` tags the last time fired value $\Psi_E(s)$.

The following is an example of an element command.

```
(Element (Time 2) (Name H) (Threshold -3) (Refractory 2) (Last 0))
```

At time 2, if active element `H` does not exist, then it is created. Active element `H` has its threshold set to -3, its refractory period set to 2, and its last time fired set to 0. After time 2, active element `H` exists indefinitely with threshold = -3, refractory = 2 until a new Element command whose name value `H` is executed at a later time; in this case, the Refractory, Threshold and Last values specified in the new command are updated.

Definition 5. `Connection`

A `Connection` command creates or updates a connection from one active element to another active element. This command has the following Backus-Naur syntax.

```
<cnct_cmd> ::= (Connection (Time <aint>)(From <ename>)(To <ename>)
                 [(Amp <int>)(Width <pos_int>)(Delay <pos_int>)] )
```

The keyword `Time` tags the time value s at which the connection is created or updated. The keyword `From` tags the name `F` of the active element that sends a pulse with these updated values. The keyword `To` tags the name `T` of the active element that receives a pulse with these updated values. The keyword `Amp` tags the pulse amplitude value $A_{FT}(s)$ that is assigned to this connection. The keyword `Width` tags the pulse width value $\omega_{FT}(s)$. The keyword `Delay` tags the transmission time $\tau_{FT}(s)$.

When the AEM clock reaches time `s`, `F` and `T` are name values that must be the name of an element that already has been created or updated before or at time s. Not all of the connection parameters need to be specified in a connection command. If the connection does not exist beforehand and the Width and Delay values are not specified appropriately, then the amplitude is set to zero and this zero connection has no effect on the AEM computation. Observe that the connection exists indefinitely with the same parameter values until a new connection is executed at a later time between `From` element `F` and `To` element `T`.

The following is an example of a connection command.

```
(Connection (Time 2) (From C) (To L) (Amp -7) (Width 1) (Delay 3))
```

At time 2, the connection from active element `C` to active element `L` has its amplitude set to -7, its pulse width set to 1, and its transmission time set to 3.

Definition 6. `Fire`
The `Fire` command has the following Backus-Naur syntax.

```
<fire_cmd> ::= (Fire (Time <aint>) (Name <ename>) )
```

The `Fire` command fires the active element indicated by the `Name` tag at the time indicated by the `Time` tag. This command is primarily used to fire input active elements in order to communicate program input to the active element machine.

An example is `(Fire (Time 3) (Name C))`, which fires active element `C` at `t`= 3.

Definition 7. `Program`
The `Program` command is convenient when a sequence of commands are used repeatedly. This command combines a sequence of commands into a single command. It has the following definition syntax.

```
<program_def> ::= (Program <pname> [(Cmds <cmds>)][(Args <args>)]
                   <cmd_sequence> )

<pname> ::= <ename>

<cmds> ::= <cmd_name> | <cmd_name><cmds>

<cmd_name> ::=  Element | Connection | Fire | Meta | <pname>

<args> ::= <symbol> | <symbol><args>
```

The `Program` command has the following execution syntax.

```
<program_cmd> ::= (<pname> [(Cmds <cmds>)] [(Args <args_cmd>)] )

<args_cmd> ::= <ename> | <ename><args_cmd>
```

The `FireN` program is an example of definition syntax.

```
(Program  FireN (Args t  E)
 (Element (Time 0) (Name E)(Refractory 1)(Threshold 1)(Last 0))
 (Connection (Time 0) (From E) (To E)(Amp 2)(Width 1)(Delay 1))
 (Fire (Time 1) (Name E))
 (Connection (Time  t+1) (From E) (To E) (Amp 0))
 )
```

The execution of the command `(FireN (Args 8 E1))` causes element `E1` to fire 8 times at times 1, 2, 3, 4, 5, 6, 7, and 8 and then `E1` stops firing at time = 9.

Definition 8. Keywords `clock` and `dT`

The keyword `clock` evaluates to an integer, which is the value of the current active element machine time. `clock` is an instance of `<ename>`. If the current AEM time is 5, then the command

```
(Element (Time clock) (Name clock) (Threshold 1) (Refractory 1)
                                                         (Last -1))
```

is executed as

```
(Element (Time 5) (Name 5) (Threshold 1) (Refractory 1) (Last -1))
```

Once command `(Element (Time clock) (Name clock) (Threshold 1) (Refractory 1) (Last -1))` is created, then at each time step this command is executed with the current time of the AEM. If this command is in the original AEM program before the clock starts at 0, then the following sequence of elements named 0, 1, 2, ... will be created.

```
(Element (Time 0) (Name 0) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 1) (Name 1) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 2) (Name 2) (Threshold 1) (Refractory 1) (Last -1))
 . . .
```

The keyword `dT` represents a positive infinitesimal amount of time. If `m` and `n` are integers and $0 \leq$ `m < n`, then `mdT < ndT`. Furthermore, `dT > 0` and `dT` is less than every positive rational number. Similarly, `-dT < 0` and `-dT` is greater than every negative rational number. The purpose of `dT` is to prevent an inconsistency in definition 1. For example, the use of `dT` helps remove the inconsistency of a `To` element about to receive a pulse from a `From` element at the same time that the connection is removed.

Definition 9. `Meta`

The `Meta` command causes a command to execute when an element fires within a window of time. This command has the following execution syntax.

```
<meta_cmd> ::= (Meta (Name <ename>) [<win_time>]  <AEM_cmd>)
<win_time> ::= (Window <aint> <aint>)
```

To understand the behavior of the Meta command, consider the execution of

```
(Meta (Name E) (Window l  w) (C (Args t a))
```

where `E` is the name of the active element. The keyword `Window` tags an interval i.e. a window of time. `l` is an integer, which locates one of the boundary points of the window of time. Usually, `w` is a positive integer, so the window of time is `[l, l+w]`. If `w` is a negative integer, then the window of time is `[l+w, l]`.

The command `C` executes each time that `E` fires during the window of time, which is either `[l, l+w]` or `[l+w, l]`, depending on the sign of `w`. If the window of time is omitted, then command `C` executes at any time that element `E` fires. In other words, effectively `l` $= -\infty$ and `w` $= \infty$.

Consider the example where the `FireN` command was defined in 7.

```
(FireN (Args 8 E1))
(Meta  (Name E1) (Window 1 5) (C (Args clock  a  b)) )
```

Command `C` is executed 6 times with arguments `clock, a, b`. The firing of `E1` *triggers* the execution of command `C`.

In regard to the Meta command, there is one assumption that is analogous to the Turing machine tape being unbounded as Turing program execution proceeds.

(See Definitions 13 and 14.) During execution of a finite active element program, an active element can fire and due to one or more Meta commands, new elements and connections can be added to the machine. As a consequence, at any time the active element machine only has a finite number of computing elements and connections but the number of elements and connections can be unbounded as a function of time as the active element program executes.

4 Resolving Concurrent Generation of AEM Commands

This section explains how to resolve concurrency issues pertaining to two or more commands about to set parameter values of the same connection or same element at the same time. Then the Fire, Meta and Program commands are covered.

For example, consider two or more connection commands, connecting the same active elements, that are generated and scheduled to execute at the same time.

```
(Connection (Time t) (From A) (To B) (Amp 2)  (Width 1) (Delay 1))
(Connection (Time t) (From A) (To B) (Amp -4) (Width 3) (Delay 7))
```

Then the simultaneous execution of these two commands can be handled by defining the outcome to be equivalent to the execution of only one connection command where the respective amplitudes, widths and transmission times are averaged.

```
(Connection (Time t) (From A) (To B) (Amp -1) (Width 2) (Delay 4))
```

In the general case, for *n* connection commands

```
(Connection (Time t) (From A) (To B) (Amp a1)(Width w1)(Delay s1))
(Connection (Time t) (From A) (To B) (Amp a2)(Width w2)(Delay s2))
. . .
(Connection (Time t) (From A) (To B) (Amp an)(Width wn)(Delay sn))
```

these commands are resolved to the execution of one connection command

```
(Connection (Time t) (From A) (To B) (Amp a) (Width w) (Delay s))
```

where `a`, `w` and `s` are defined based on the application. For theoretical studies of the AEM, averaging the respective amplitudes, widths and transmission times can be useful in mathematical proofs.

```
  a = (a1 + a2 +  . . .  + an) / n
  w = (w1 + w2 +  . . .  + wn) / n
  s = (s1 + s2 +  . . .  + sn) / n
```

For some applications, when there is noisy environmental data fed to the input elements and amplitudes, widths and transmission times are evolved and mutated ([9], [14], [17], [19], [26], [27], [29]), extremely large (in absolute value) amplitudes, widths and transmission times can arise that skew an average function. In this context, computing the median of the amplitudes, widths and delays provides a simple method to address skewed amplitude, width and transmission time values.

```
a = median(a1, a2,  . . . , an)
w = median(w1, w2,  . . . , wn)
s = median(s1, s2,  . . . , sn)
```

Another alternative is to add the parameter values.

```
a = a1 + a2 +  . . . + an
w = w1 + w2 +  . . . + wn
s = s1 + s2 +  . . . + sn
```

Similarly, consider when two or more element commands – that all specify the same active element `E` – are generated and scheduled to execute at the same time.

```
(Element (Time t) (Name E)(Threshold h1)(Refractory r1)(Last s1))
(Element (Time t) (Name E)(Threshold h2)(Refractory r2)(Last s2))
 . . .
(Element (Time t) (Name E)(Threshold hn)(Refractory rn)(Last sn))
```

Resolve these commands to the execution of one element command,

```
(Element (Time t) (Name E) (Threshold h) (Refractory r) (Last s))
```

where `h`, `r` and `s` are defined based on the application. Similar to the connection command, for theoretical studies of the AEM, the threshold, refractory and last time fired values can be averaged.

```
h = (h1 + h2 + . . . + hn) / n
r = (r1 + r2 + . . . + rn) / n
s = (s1 + s2 + . . . + sn) / n
```

In autonomous applications, where evolution of parameter values occurs, the median can also help address skewed values in the element commands.

```
h = median(h1, h2, . . ., hn)
r = median(r1, r2, . . ., rn)
s = median(s1, s2, . . ., sn)
```

Another alternative is to add the parameter values.

```
h = h1 + h2 + . . . + hn
r = r1 + r2 + . . . + rn
s = s1 + s2 + . . . + sn
```

Rules 1, 2, and 3 resolve concurrency issues pertaining to the Fire, Meta and Program commands.

1. If two or more Fire commands attempt to fire element `E` at time `t`, then element `E` is fired just once at time `t`.
2. Only one Meta command can be triggered by the firing of an active element. If a new Meta command is created and it happens to be triggered by the same element `E` as a prior Meta command, then the old Meta command is removed and the new Meta command is triggered by element `E`.

3. If a Program command is called by a Meta command, then the Program's internal Element, Connection, Fire and Meta commands follow the previous concurrency rules defined. If a Program command exists within a Program command, then these rules are followed recursively on the nested Program command.

5 Copy and Nand Program Examples

These examples demonstrate an active element copy program and a nand program.

Example 2. *Copy Program*
This active element program copies an element's firing state to another element.

```
(Program copy (Args s t b a)
(Element (Time s-1)(Name b)(Threshold 1)(Refractory 1)(Last s-1))
(Connection (Time s-1)(From a) (To b)(Amp 0) (Width 0) (Delay 1))
(Connection (Time s) (From a) (To b) (Amp 2) (Width 1) (Delay 1))
(Connection (Time s) (From b) (To b) (Amp 2) (Width 1) (Delay 1))
(Connection (Time t) (From a) (To b) (Amp 0) (Width 0) (Delay 1))
)
```

When the `copy` program is called, active element `b` will start firing if `a` fired during the window of time [`s`, `t`). Further, a connection is set up from `b` to `b` so that `b` will keep firing indefinitely. This enables `b` to *store* active element `a`'s firing state.

Example 3. *Nand Program*
This active element program computes a nand circuit.

```
(Program nand (Args s x y l h)
(Element (Time s) (Name x) (Threshold 0) (Refractory 1) (Last s))
(Element (Time s) (Name y) (Threshold 0) (Refractory 1) (Last s))
(Element (Time s) (Name h) (Threshold -3)(Refractory 2) (Last s))
(Element (Time s) (Name l) (Threshold 3) (Refractory 2) (Last s))
(Connection (Time s) (From x) (To l) (Amp 2) (Width 1) (Delay 1))
(Connection (Time s) (From x) (To h) (Amp -2)(Width 1) (Delay 1))
(Connection (Time s) (From y) (To l) (Amp 2) (Width 1) (Delay 1))
(Connection (Time s) (From y) (To h) (Amp -2)(Width 1) (Delay 1))
)
```

`B` and `C` are input elements. `L` (represents a 0 output) and `H` (represents a 1 output) are output elements. The call `(nand (Args -1 B C L H))` creates the connections between `B`, `C`, `L` and `H`. All four cases are verified where $|$ denotes the Sheffer stroke.

1. At $t = 0$, active elements `B` and `C` do not fire i.e. `B` $|$ `C` $= 0|0 = 1$. Since the threshold of `H` is -3, `H` fires at time $t = 1$.
2. At $t = 0$, active element `B` fires and `C` does not fire i.e. `B` $|$ `C` $= 1|0 = 1$. Since one pulse with amplitude -2 reaches `H` at $t = 1$ just as the refractory period expires and $-2 > -3$, then `H` fires at time t = 1.

3. At $t = 0$, active element `B` does not fire and `C` fires i.e. `B` $\mid$ `C` $= 0|1 = 1$. Since one pulse with amplitude -2 reaches `H` at $t = 1$ just as the refractory period expires and $-2 > -3$, then `H` fires at time t = 1
4. At $t = 0$, active elements `B` and `C` both fire i.e. `B` $\mid$ `C` $= 1|1 = 0$. At time $t = 1$, two pulses with amplitude -2 reach `H` so `H` does not fire. At time $t = 1$, two pulses reach `L`, and each pulse has amplitude 2. `L` has threshold $= 3$, so `L` fires at $t = 1$.

6 Machine Computation

The `nand` program example uses the firing of output elements `L`, `H` to represent the low, high outputs respectively. The firing of elements was used to represent the computation of a boolean function. In the next set of definitions, firing representations, machine computation and interpretation are formalized.

Definition 10. *Firing Representation*
Consider active element E_i's firing times in the interval of time $W = [t_1, t_2]$. Let s_1 be the earliest firing time of E_i lying in W, and s_n the latest firing time lying in W. Then E_i 's firing sequence $F(E_i, W) = [s_1, \ldots, s_n] = \{s \in W : g_i(s) = 1\}$ is called a firing sequence of the active element E_i over the window of time W. From active elements $\{E_1, E_2, \ldots, E_n\}$, create the tuple $(F(E_1, W), F(E_2, W), \ldots, F(E_n, W))$, which is called a firing representation of the active elements $\{E_1, \ldots, E_n\}$ within the window of time W.

At the machine level of interpretation, firing representations express the input to, the computation of, and the output of an active element machine. At a more abstract level, firing representations can represent an input symbol, an output symbol, a sequence of symbols, a spatio-temporal pattern, a number, or even a sequence of program instructions.

Definition 11. *Sequence of Firing Representations*
Let $W_1, W_2, \ldots, W_n$ be a sequence of time intervals. Let $\mathscr{F}(\mathscr{E}, W_1) = (F(E_1, W_1), F(E_2, W_1), \ldots, F(E_n, W_1))$ be a firing representation of active elements $\mathscr{E} = \{E_1, E_2, \ldots, E_n\}$ over the interval W_1. In general, let $\mathscr{F}(\mathscr{E}, W_k) = (F(E_1, W_k), F(E_2, W_k), \ldots F(E_n, W_k))$ be a firing representation over the interval of time W_k. From these, a sequence of firing representations, $[\mathscr{F}(\mathscr{E}, W_1), \mathscr{F}(\mathscr{E}, W_2), \ldots, \mathscr{F}(\mathscr{E}, W_n)]$ is created.

Definition 12. *Machine Computation*
Let $[\mathscr{F}(\mathscr{E}, W_1), \mathscr{F}(\mathscr{E}, W_2), \ldots, \mathscr{F}(\mathscr{E}, W_n)]$ be a sequence of firing representations. $[\mathscr{F}(\mathscr{E}, S_1), \mathscr{F}(\mathscr{E}, S_2), \ldots, \mathscr{F}(\mathscr{E}, S_m)]$ is some other sequence of firing representations. Suppose machine architecture $\mathscr{M}(\mathscr{I}, \mathscr{E}, \mathscr{O})$ has input active elements $\mathscr{I}$ fire with the pattern $[\mathscr{F}(\mathscr{E}, S_1), \mathscr{F}(\mathscr{E}, S_2), \ldots, \mathscr{F}(\mathscr{E}, S_m)]$ and consequently $\mathscr{M}$'s output active elements $\mathscr{O}$ fire according to $[\mathscr{F}(\mathscr{E}, W_1), \mathscr{F}(\mathscr{E}, W_2), \ldots, \mathscr{F}(\mathscr{E}, W_n)]$. In this case, the machine $\mathscr{M}$ *computes* $[\mathscr{F}(\mathscr{E}, W_1), \mathscr{F}(\mathscr{E}, W_2), \ldots, \mathscr{F}(\mathscr{E}, W_n)]$ from $[\mathscr{F}(\mathscr{E}, S_1), \mathscr{F}(\mathscr{E}, S_2), \ldots, \mathscr{F}(\mathscr{E}, S_m)]$.

An active element machine is an *interpretation* between two sequences of firing representations if the machine can compute the output sequence of firing representations from the input sequence of firing representations. Using the definition of machine computation, examples 2 and 3 help derive the following theorem.

Theorem 1. *A register machine with an unbounded number of registers can be constructed with an active element program.*

Proof. A proof is sketched. A finite boolean function can be constructed by composing a finite number of nand circuits. The repeated use of the copy program enables an active element machine to store an unbounded amount of state in terms of bits. A register machine can be constructed from the boolean functions and the ability to store state. An unbounded number of registers can be supported by the active element machine because the Element and Connection commands enable the program to add new Elements and Connections at any time. Thus, new registers can be added as needed during the computation of the register machine program.

Corollary 1. *Any Turing computable function can be computed by some active element machine, specified by a finite active element program.*

Proof. In [44], they show that the partial recursive functions are the same as the functions computable by their register machine model and consequently the same as the Turing computable functions (see [45], and Definitions 13, 14, 15 and 16). The corollary follows from this fact and Theorem 1.

Example 4. Randomness generates an AEM, representing a real number in $[0,1]$
Using a random process in the environment to fire or not fire one input element `I` at each unit of time, an active element program is demonstrated with a firing representation of an arbitrary real number in the unit interval $[0,1]$.

This example uses a random process from the environment to either fire input element `I` or not fire `I` at time $t = n$ where n is a natural number $\{0,1,2,3,\ldots\}$. This random sequence of 0 and 1's can be generated by quantum optics ([28], [43]) or another type of quantum or physical phenomena [2].

Using the Meta command, the random sequence of bits creates active elements $0,1,2,\ldots$ that store the binary representation $b_0b_1b_2\ldots$ of real number $x \in [0,1]$. If input element `I` fires at time $t = n$, then $b_n = 1$; thus, create active element `n` so that after $t = n$, element `n` fires every unit of time indefinitely. If input element `I` does not fire at time $t = n$, then $b_n = 0$ and active element `n` is created so that it never fires. The following finite active element machine program exhibits this behavior.

```
(Program C  (Args t)
(Connection (Time t) (From I) (To t) (Amp 2) (Width 1) (Delay 1))
(Connection (Time t+1+dT) (From I) (To t) (Amp 0))
(Connection (Time t) (From t) (To t) (Amp 2) (Width 1) (Delay 1))
)
(Element (Time clock) (Name clock) (Threshold 1)  (Refractory 1)
                                                   (Last -1))
```

```
(Meta (Name I) (C (Args clock)))
```

Next, this program is explained, assuming the sequence of random bits from the environment begins with 1, 0, 1, ... Thus, input element `I` fires at times 0, 2, ... At time 0, the following commands are executed.

```
(Element (Time 0) (Name 0) (Threshold 1) (Refractory 1)(Last -1))
(C (Args 0))
```

The execution of `(C (Args 0))` causes the three connection commands to execute.

```
(Connection (Time 0) (From I) (To 0) (Amp 2) (Width 1) (Delay 1))
(Connection (Time 1+dT) (From I) (To 0) (Amp 0))
(Connection (Time 0) (From 0) (To 0) (Amp 2) (Width 1) (Delay 1))
```

Because of the first connection command

```
(Connection (Time 0) (From I) (To 0) (Amp 2) (Width 1) (Delay 1))
```

the firing of input element `I` at time 0 sends a pulse with amplitude 2 to element `0`. Thus, element `0` fires at time 1. Then at time `1+dT`, a moment after time 1, the connection from input element `I` to element `0` is removed. At time 0, a connection from element `0` to itself with amplitude 2 is created. As a result, element `0` continues to fire indefinitely, representing that $b_0 = 1$.

At time 1, command

```
(Element (Time 1) (Name 1) (Threshold 1) (Refractory 1)(Last -1))
```

is created. Since element `1` has no connections into it and threshold 1, element `1` never fires. Thus $b_1 = 0$.

At time 2, input element `I` fires, so the following commands are executed.

```
(Element (Time 2) (Name 2) (Threshold 1) (Refractory 1)(Last -1))
(C (Args 2))
```

The execution of `(C (Args 2))` causes the three connection commands to execute.

```
(Connection (Time 2) (From I) (To 2) (Amp 2) (Width 1) (Delay 1))
(Connection (Time 3+dT) (From I) (To 2) (Amp 0))
(Connection (Time 2) (From 2) (To 2) (Amp 2) (Width 1) (Delay 1))
```

Because of the first connection command

```
(Connection (Time 2) (From I) (To 2) (Amp 2) (Width 1) (Delay 1))
```

the firing of input element `I` at time 2 sends a pulse with amplitude 2 to element `2`. Thus, element `2` fires at time 3. Then at time `3+dT`, a moment after time 3, the connection from input element `I` to element `2` is removed. At time 2, a connection from element `2` to itself with amplitude 2 is created. As a result, element `2` continues to fire indefinitely, representing that $b_2 = 1$.

7 AEM Binary Language Recognizer

Based on example 4, this section shows how to build a binary language recognizer, but do not focus on optimizing the speed of the binary language recognition. The next section demonstrates active elements performing simultaneous computations on an NP-hard problem.

First, some definitions and notation for binary strings and languages are reviewed. A binary string is a finite sequence of 0s and 1s. The set of binary strings of length n is denoted as $\{0,1\}^n$. If string $w \in \{0,1\}^k$, then w's length is $|w| = k$. The set of all finite binary strings is denoted as $\{0,1\}^* = \bigcup_{n=1}^{\infty}\{0,1\}^n$. Consider function $f : \{0,1\}^* \to \{0,1\}$, then $L = f^{-1}(1)$ is a binary language. Every binary language can be represented as the inverse image $g^{-1}(1)$ for some unique function $g : \{0,1\}^* \to \{0,1\}$. For any function $f : \{0,1\}^* \to \{0,1\}$, define $f_n : \{0,1\}^n \to \{0,1\}$ such that f restricted on $\{0,1\}^n = f_n$ i.e. $f_n(u) = f(u)$ for all u in $\{0,1\}^n$.

An ordering of the strings $[\{0,1\}^*, \prec]$ is created. For any two distinct strings $u, w \in \{0,1\}^*$, then $u \prec w$ if $|u| < |w|$; $w \prec u$ if $|w| < |u|$. Otherwise, $|u| = |w|$ in which case $u \prec w$ if u is smaller than w, treating u, w as binary numbers; or $w \prec u$ if w is smaller than u as binary numbers.

There are 2^n binary strings of length n. Let $\phi(n) = 2^n$ and $\beta(n) = \phi(\phi(n))$. From the following diagram, it is an easy observation

$$\begin{array}{ccc} 000\ldots0 & \longrightarrow & 0 \text{ or } 1 \\ 100\ldots0 & \longrightarrow & 0 \text{ or } 1 \\ 010\ldots0 & \longrightarrow & 0 \text{ or } 1 \\ 110\ldots0 & \longrightarrow & 0 \text{ or } 1 \\ & \ldots & \\ 111\ldots1 & \longrightarrow & 0 \text{ or } 1 \end{array}$$

that there are $\beta(n)$ distinct Boolean functions $f_n : \{0,1\}^n \to \{0,1\}$. Each integer c_n satisfying $0 \le c_n < \beta(n)$ corresponds to a distinct $f_n : \{0,1\}^n \to \{0,1\}$.

This correspondence can be implemented with an active element machine that uses the elements created in example 4 to encode for the numbers $c_1, c_2, \ldots$. The encoding of boolean function f_1 is stored (represented) in the first $\beta(1)$ active elements 0, 1, 2, 3, which represent natural number c_1. For each natural number n, the boolean function f_n is represented with the next $\beta(n)$ elements generated by the Meta command from the random input to element `I`, as described in example 4.

These groups of active elements representing c_1, c_2, ... are called registers. In general, register R_{-n} stores c_n as shown in table 3.

Table 3 Register encoding of Boolean functions f_n and binary string $a_1 \ldots a_m$

Register	...	$-n$	...	-1	0	1	...	m
Contents	...	c_n	...	c_1	m	a_1	...	a_m

The binary string of length m that is accepted or not accepted by this machine is stored in registers $R_1, R_2, \ldots, R_m$ and represented as $a_1 a_2 \ldots a_m$ where each a_k is a 0 or 1. Each register R_k can be represented with a single active element J_k where it fires indefinitely if $a_k = 1$ and never fires if $a_k = 0$.

In order to refer to the register of a register, use the notation R_{R_n}. If register R_n contains -5 and the contents of $R_{-5} = 29$, then the contents of $R_{R_n} = 29$.

Solution 1. *Binary Language Decision Steps 1 to 6*
Steps 1 to 6 decide whether a binary string $S = a_1 a_2 \dots a_m$ is in the language determined by the active element machine. Before step 1 is started, for each natural number $n \leq m$, the value c_n is stored in register R_{-n}, where $0 \leq c_n < \beta(n)$. Since each c_n uses 2^n bits to store c_n, input element `I` needs $1 + 2 + \dots 2^m$ time steps i.e. $1 + 2 + \dots 2^m$ random bits from the environment before all these register values are assigned.

The values in registers R_1, R_2, $\dots$, R_m are 0 or 1 where R_k, such that $1 \leq k \leq m$, stores the value of a_k. The value m is stored in register 0.

```
1. Read the string length $m$ from register $R_0$. Compute $2^m$. Store in register $R_{m+1}$.
2. Store 0 in register $R_{m+2}$. Store $c_m$ in register $R_{m+3}$.
3. Initialize registers $R_{m+5}$, $\dots$, $R_{m+4+2^m}$ with 0.
4. In a loop that is executed $c_m$ times,
   Do
   {
      Increment register $R_{m+2}$.
      Increment the binary number in registers $R_{m+5}, \dots R_{m+4+2^m}$.
   } Until register $R_{m+2}$ equals register $R_{m+3}$.

5. Compute $(m + 5 +$ binary $a_1 a_2 \dots a_m)$ stored in $R_1$, $R_2$, $\dots$, $R_m$.
   Store 1 in register $R_{m+1}$.
   Store $m + 5$ in register $R_{m+2}$.
   Store 0 in register $R_{m+3}$.
   Store 1 in register $R_{m+4}$.
   In a loop that is executed $m$ times,
   Do
   {
      If the contents of $R_{R_{m+1}}$ equals 1, add the value in $R_{m+4}$ to $R_{m+2}$.
      Increment $R_{m+3}$.
      Increment $R_{m+1}$.
      Double value in $R_{m+4}$ and store back in register $R_{m+4}$.
      } Until the value in $R_{m+3}$ equals $m$.

6. If register $R_{R_{m+2}}$ contains a 1 then binary string $a_1 a_2 \dots a_m$ is in the language
   recognized by this machine. Otherwise, $R_{R_{m+2}}$ contains a 0 and binary string
   $a_1 a_2 \dots a_m$ is not in the language recognized by this machine.
```

Example 5. *Steps 1 through 6 when* $c_3 = 203$
This example shows the encoding of f_3 when $c_3 = 203$ and describes steps 1, 2, ..., 6 as they are executed.

Suppose R_{-3} has $c_3 = 203$ stored in it. This example demonstrates how $c_3 = 203$ encodes for the corresponding f_3 and how $f_3(S)$ is computed for any S in $\{0,1\}^3$. Decimal number 203 in reversed binary is $11010011 = 2^0 + 2^1 + 2^3 + 2^6 + 2^7$. The ordering of $\{0,1\}^3 = [000, 100, 010, 110, 001, 101, 011, 111]$, which is iterated in this order in the loop of step 5. The bits of 1101 0011 represent the range values of f_3 where $f_3(000)$ = the 1st bit in 1101 0011 ; $f_3(100)$ = the 2nd bit in 1101 0011 ; . . . ; $f_3(111)$ = the eighth (2^3) bit in 1101 0011. This is summarized in table 4.

Table 4 Boolean functions f_3

$f_3(000) = 1$	$f_3(100) = 1$	$f_3(010) = 0$	$f_3(110) = 1$
$f_3(001) = 0$	$f_3(101) = 0$	$f_3(011) = 1$	$f_3(111) = 1$

Before program execution starts the contents of $R_0 = 3$, $R_1 = 0$, $R_2 = 1$ and $R_3 = 1$. Thus, the program will decide whether $S = 011$ lies in the language determined by this machine. Steps 1 to 6 execute as follows.

- Step 1 reads 3 from R_0 and computes $2^3 = 8$. Then 8 is stored in R_4.
- Step 2 stores the contents of $R_{-R_0} = 203$ in register R_6. Also, 0 is stored in R_5.
- Step 3 stores 0 in R_8, R_9 , $R_{10}, \ldots, R_{15}$.
- Step 4 executes the loop 203 times. The binary counter increments registers R_8, R_9, $R_{10}, \ldots, R_{15}$ in reversed binary representation. After executing this loop 203 times, $R_8 = 1$, $R_9 = 1$, $R_{10} = 0$, $R_{11} = 1$, $R_{12} = 0$, $R_{13} = 0$, $R_{14} = 1$, $R_{15} = 1$.
- Step 5 initializes R_{m+2} to 8. The loop adds binary number R_1, R_2, R_3 to 8 and stores it in R_{m+2}. The purpose of register R_{m+2} is that it indexes the value of $f_3(011)$ in step 6 which decides whether 011 is in the language computed by this machine. After the first pass through the loop, since $R_1 = 0$, then $R_{m+2} = 8$. After the second pass, $R_{m+4} = 2$, so $R_{m+2} = 10$. After the third and final pass through the loop, $R_{m+4} = 4$, so $R_{m+2} = 14$.
- In Step 6, the value of $R_{m+2} = 14$. Thus, $R_{R_{m+2}} = R_{14} = 1$. Thus, $f_3(011) = 1$, so 011 is in the language computed by this machine.

Storing and initializing can be implemented with a register machine; incrementing can be implemented with a register machine; looping, doubling, counting and adding can be implemented with a register machine; and computing 2^m involves executing a doubling routine inside a loop that is executed m times. From these observations and Theorem 1, steps 1 through 6 can be implemented with an active element machine.

Next, three lemmas and a theorem are presented. These results prove that steps 1 through 6 decide whether an arbitrary binary string $S = a_1 a_2 \dots a_m$ lies in the language determined by this active element machine.

Lemma 1. *For any $L \subseteq \{0,1\}^*$, then $L = f^{-1}\{1\}$ for some $f : \{0,1\}^* \to \{0,1\}$.*

Proof. Set $L_n = L \cap \{0,1\}^n$. Then $L_n \cap L_m = \emptyset$ whenever $n \neq m$. Define the boolean function $f_n : \{0,1\}^n \to \{0,1\}$ as follows. For each S in $\{0,1\}^n$, if S lies in L, then define $f_n(S) = 1$. If S does not lie in L, then define $f_n(S) = 0$. Set $f = \cup_{n=1}^{\infty} f_n$. Then $f^{-1}\{1\} = L$.

Lemma 2. *For each c_n, satisfying $0 \le c_n < \beta(n)$ and such that R_{-n} contains c_n before program execution starts, then for any binary string $S = a_1 a_2 \dots a_n$ in $\{0,1\}^n$, step 6 decides whether S lies in the language of the machine. As S ranges over each element of $\{0,1\}^n$, then this determines the set $L_n \subseteq \{0,1\}^n$ of all binary strings of length n, that are recognized by this machine.*

Proof. Step 4 is iterated c_n times where $0 \le c_n < \beta(n)$ and each time through the loop, the contents of registers $R_{n+5} \dots, R_{n+4+2^n}$ are incremented as a binary number where R_{n+5} stores the 2^0 bit, R_{n+6} stores the 2^1 bit, . . ., and R_{n+4+2^n} stores the 2^n bit. As a result, step 4 creates a sequence of 0's and 1's, representing the binary encoding of f_n, according to the ordering of $\{0,1\}^n$ created in step 5. Step 6 uses this encoding and ordering to determine whether a binary string S of length n is recognized by the machine.

Lemma 3. *Steps 1 through 6 with c_n initially stored in R_{-n}, where $0 \le c_n < \beta(n)$ before program execution starts, decide a unique language $L \subseteq \{0,1\}^n$. If $b_n \neq c_n$ and $0 \le b_n, c_n < \beta(n)$, then the languages determined by these two different values are distinct.*

Proof. Let $K_n \subseteq \{0,1\}^n$ be the language decided by machine K with register R_{-n} containing b_n. Let $L_n \subseteq \{0,1\}^n$ be the language decided by machine L with register R_{-n} containing c_n. When a binary string of length n is stored in registers $R_1, R_2, \dots, R_n$ before program execution, then after step 4 is completed, the values of whether to accept or not accept a binary string of length n are stored as $R_{n+5}, \dots,$ R_{n+4+2^n}. If b_n stored in register R_{-n} of machine K is not equal to c_n stored in R_{-n} of machine L, then after step 4 is completed R_j in machine K is not equal to R_j in machine L for some j satisfying $n + 5 \le j \le n + 4 + 2^n$. This implies that language $K_n \neq L_n$.

Theorem 2. *There is a one to one correspondence between the binary languages $L \subseteq \{0,1\}^*$ and the sequence of natural numbers $c_1, c_2, \dots, c_n, \dots,$ such that for each n, the natural number c_n satisfies $0 \le c_n < \beta(n)$. Furthermore, if each register R_{-n} is initialized to c_n where $1 \le n \le m$ before program execution of step 1 begins, then the program execution recognizes this corresponding binary language, where the binary string of length m to be recognized is initially stored in registers $R_1, R_2, \dots, R_m$.*

Proof. This theorem follows immediately from the three previous lemmas.

8 An AEM Program Computes a Ramsey Number

This section shows how to compute a Ramsey number with an AEM program. Ramsey theory can be intuitively described as the study of structure which is preserved under finite decomposition (see [10], [22], [36]). Applications of Ramsey theory include results in number theory [40], algebra, geometry [15], topology [20], set theory, logic [36], ergodic theory [20], computer science ([3], [4], [5], [6], [7], [8], [46]), including lower bounds for parallel sorting [16], game theory [24] and information theory ([37], [41]). Progress on determining the basic Ramsey numbers $r(k,l)$ has been slow. For positive integers k and l, $r(k,l)$ denotes the least integer n such that if the edges of the complete graph K_n are 2-colored with colors red and blue, then there always exists a complete subgraph K_k containing all red edges or there exists a subgraph K_l containing all blue edges.

To put our slow progress into perspective, arguably the best combinatorist of the 20th century, Paul Erdös asks us to imagine an alien force, vastly more powerful than us, landing on Earth and demanding the value of $r(5,5)$ or they will destroy our planet. In this case, Erdös claims that we should marshal all our computers and all our mathematicians and attempt to find the value. But suppose instead that they ask for $r(6,6)$. For $r(6,6)$, Erdös believes that we should attempt to destroy the aliens [42].

Theorem 3. *The standard finite Ramsey theorem.*

For any positive integers m,k,n, there is a least integer $N(m,k,n)$ with the following property: no matter how one colors each of the n-element subsets of $S = \{1,2,...,N\}$ with one of k colors, there exists a subset Y of S with at least m elements, such that all n-element subsets of Y have the same color *(See [23], [36], [39]).*

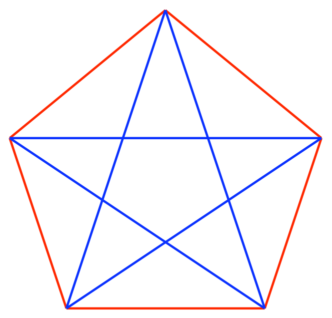
When G and H are simple graphs, there is a special case of theorem 3. Define the Ramsey number $r(G,H)$ to be the smallest N such that if the complete graph K_N is colored red and blue, either the red subgraph contains G or the blue subgraph contains H. (A simple graph is an unweighted, undirected graph containing no graph loops or multiple edges. In a simple graph, the edges of the graph form a set and each edge is a pair of distinct vertices.) In [10], S.A. Burr proves that determining $r(G,H)$ is an NP-hard problem.

An AEM program is designed that solves a special case of Theorem 3. Color each edge of the complete graph K_6 red or blue. Then there is always at least one triangle, which contains only blue edges or only red edges. In terms of the standard Ramsey theorem, this is the special case $N(3,2,2)$ where $n = 2$ since edges are colored (i.e. 2-element subsets); $k = 2$ since two colors are used; and $m = 3$ since one seeks a red or blue triangle.

To demonstrate how an AEM program can be designed to compute $N(3,2,2) = 6$, an AEM program is designed that verifies $N(3,2,2) > 5$, based on figure 1.

The symbols `B` and `R` represent blue and red, respectively. Place indices on `B` and `R` to denote active elements that correspond to the K_5 graph geometry. First, the indices are derived from the graph geometry. Let $E = \{\{1,2\},\{1,3\},\{1,4\},\{1,5\},$

Fig. 1 A 2-coloring of K_5 containing no monochromatic K_3 (triangle)



$\{2,3\}, \{2,4\}, \{2,5\}, \{3,4\}, \{3,5\}, \{4,5\}\}$ denote the edge set of K_5. The triangle set $T = \{\{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,3,4\}, \{1,3,5\}, \{1,4,5\}, \{2,3,4\}, \{2,3,5\}, \{2,4,5\}, \{3,4,5\}\}$. As shown in figure 1, each edge is colored red or blue. Thus the red edges are $\{\{1,2\}, \{1,5\}, \{2,3\}, \{3,4\}, \{4,5\}\}$ and the blue edges are $\{\{1,3\}, \{1,4\}, \{2,4\}, \{2,5\}, \{3,5\}\}$.

Number each group of AEM commands for K_5, based on the group's purpose. This is useful because these groups are referred to when describing the computation for K_6.

1. The elements representing red and blue edges are established as follows.

```
(Element (Time 0) (Name R_12) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name R_15) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name R_23) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name R_34) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name R_45) (Threshold 1) (Refractory 1) (Last -1))

(Element (Time 0) (Name B_13) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name B_14) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name B_24) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name B_25) (Threshold 1) (Refractory 1) (Last -1))
(Element (Time 0) (Name B_35) (Threshold 1) (Refractory 1) (Last -1))
```

2. Fire element `R_jk` if edge $\{j,k\}$ is red.

```
(Fire (Time 0) (Name R_12))
(Fire (Time 0) (Name R_15))
(Fire (Time 0) (Name R_23))
(Fire (Time 0) (Name R_34))
(Fire (Time 0) (Name R_45))
```

   Fire element `B_jk` if edge $\{j,k\}$ is blue where $j < k$.

```
(Fire (Time 0) (Name B_13))
(Fire (Time 0) (Name B_14))
(Fire (Time 0) (Name B_24))
(Fire (Time 0) (Name B_25))
(Fire (Time 0) (Name B_35))
```

3. The following Meta commands cause these elements to keep firing after they have fired once.

```
(Meta (Name  R_jk) (Window 0 1)
 (Connection (Time 0) (From R_jk) (To R_jk) (Amp 2) (Width 1) (Delay 1)))

(Meta (Name  B_jk) (Window 0 1)
 (Connection (Time 0) (From B_jk) (To B_jk) (Amp 2) (Width 1) (Delay 1)))
```

4. To determine if a blue triangle exists on vertices $\{i, j, k\}$, where $\{i, j, k\}$ ranges over T, three connections are created for each potential blue triangle.

```
(Connection (Time 0) (From B_ij) (To B_ijk) (Amp 2) (Width 1) (Delay 1))
(Connection (Time 0) (From B_jk) (To B_ijk) (Amp 2) (Width 1) (Delay 1))
(Connection (Time 0) (From B_ik) (To B_ijk) (Amp 2) (Width 1) (Delay 1))
```

5. To determine if a red triangle exists on vertex set $\{i, j, k\}$, where $\{i, j, k\}$ ranges over T, three connections are created for each potential red triangle.

```
(Connection (Time 0) (From R_ij) (To R_ijk) (Amp 2) (Width 1) (Delay 1))
(Connection (Time 0) (From R_jk) (To R_ijk) (Amp 2) (Width 1) (Delay 1))
(Connection (Time 0) (From R_ik) (To R_ijk) (Amp 2) (Width 1) (Delay 1))
```

6. For each vertex set $\{i, j, k\}$ in T, the following elements are created.

```
(Element (Time 0) (Name R_ijk) (Threshold 5) (Refractory 1) (Last -1))
(Element (Time 0) (Name B_ijk) (Threshold 5) (Refractory 1) (Last -1))
```

Because the threshold is 5, the element `R_ijk` only fires when all three elements `R_ij`, `R_jk`, `R_ik` fired one unit of time ago. Likewise, the element `B_ijk` only fires when all three elements `B_ij`, `B_jk`, `B_ik` fired one unit of time ago. From this, observe that as of `clock` $= 3$ i.e. 4 time steps, this AEM program determines that $N(3,2,2) > 5$. This AEM computation uses $|E| + 2|T| = \frac{5!}{2!3!} + 2\frac{5!}{3!2!} = 30$ active elements. Further, this AEM program creates and uses $3|T| + 3|T| + |E| = 70$ connections.

For K_6, the edge set $E = \{\{1,2\}, \{1,3\}, \{1,4\}, \{1,5\}, \{1,6\}, \{2,3\}, \{2,4\}, \{2,5\}, \{2,6\}, \{3,4\}, \{3,5\}, \{3,6\}, \{4,5\}, \{4,6\}, \{5,6\}\}$. The triangle set $T = \{ \{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,2,6\}, \{1,3,4\}, \{1,3,5\}, \{1,3,6\}, \{1,4,5\}, \{1,4,6\}, \{1,5,6\}, \{2,3,4\}, \{2,3,5\}, \{2,3,6\}, \{2,4,5\}, \{2,4,6\}, \{2,5,6\}, \{3,4,5\}, \{3,4,6\}, \{3,5,6\}, \{4,5,6\}\}$. For each 2-coloring of E, each edge is colored red or blue. There are $2^{|E|}$ 2-colorings of E. For this graph, $|E| = \frac{6!}{2!4!}$.

To build a similar AEM program, the commands in groups 1 and 2 range over every possible 2-coloring of E. The remaining groups 3, 4, 5 and 6 are the same based on the AEM commands created in groups 1 and 2 for each particular 2-coloring.

This AEM program verifies that every 2-coloring of E contains at least one red triangle or one blue triangle i.e. $N(3,2,2) = 6$. There are no optimizations using graph isomorphisms [32]. If one builds an AEM language construct for generating all active elements for each 2-coloring of E at time zero, then the resulting AEM program can determine the answer in 5 time steps. (The program needs one more time step, 2^{15} additional connections and one additional element to verify that every one of the 2^{15} AEM programs is indicating that it found a red or blue triangle.) This AEM program – that determines the answer in 5 time steps – uses $2^{|E|}(|E| + 2|T|) + 1$ active elements and $2^{|E|}(3|T| + 3|T| + |E| + 1)$ connections, where $|E| = 15$ and $|T| = 20$.

9 Discussion and Further Work

First, the results in section 7 are summarized. It is well-known ([13], [33], [45]) that there are an uncountable number of binary languages in $\{0,1\}^*$, for which no standard Turing machine with a finite number of non-blank symbols on the tape is able to recognize. This is also true for a finite register machine program ([13], [33], [44]). Yet by building upon example 4, using randomness from the environment and the Meta command, an AEM is generated with a finite number of AEM commands, that is able to recognize an arbitrary language $L \subseteq \{0,1\}^*$. This suggests that evolutionary methods ([9], [14], [17], [26], [27], [29]) along with quantum randomness ([11]) may be able to implicitly design AEMs that can exhibit useful computing behavior, which explicit register machine programs are unable to attain.

In regard to section 8, an extension of the Ramsey theorem (large Ramsey numbers) occurs when the set Y is large. A set Y is large if its cardinality is larger than its smallest element (e.g. $Y = \{1,2,3\}$). Large Ramsey numbers $L(m,k,n)$ grow too fast to be provably total in Peano arithmetic [34]. As a consequence of the extremely high growth rate of large Ramsey numbers, studying the computation of these numbers with AEM programs might be useful to help understand how to best use parallelism, geometry and time. Another area of interest is the trade-off of using a separate AEM to find graph isomorphisms that eliminate isomorphic 2-colorings, when testing for a red or blue triangle [32]. An additional area to explore would use AEM computations that rely on the simple property that every subgraph, of a monochromatic graph G, is monochromatic.

Acknowledgements Words can not express my depth of gratitude and appreciation for my wonderful wife Joanne Gomez and our daughter Haley, who has been a constant source of inspiration. I would like to thank Michael Jones, Alan Langman, David Lewis, Alex Mayer, Lutz Mueller, Don Saari and Bruce Tow for their helpful advice. In addition, many thanks to Wolfgang Halang, Kyandoghere Kyamakya, Francesca Saglietti, Jean Chamberlain Chedjou, Gerd Karl Heinz, Alireza Fasih and many others for their helpful advice at the 2011 Autonomous Systems conference.

Appendix

Below is a Turing Machine definition, where the program definition η is explicitly represented as a function instead of quintuples ([13], [45]).

Definition 13. *Turing Machine*
A Turing machine is a triple (Q, A, η) where

- Q is a finite set of states that does not contain the halt state. The states are sometimes represented as natural numbers $Q = \{2, ..., K\}$. There is a unique halt state, represented as h or as 1.
- When machine execution begins, the machine is in an initial state s and $s \in Q$.

- A is a finite set of alphabet symbols that are read from and written to the tape.
- -1 and $+1$ represent advancing the tape head to the left or right square, respectively.
- η is a function where $\eta : Q \times A \to (Q \cup \{h\}) \times A \times \{-1, +1\}$. η acts as the program for the Turing machine. For each q in Q and α in A, $\eta(q, \alpha) = (r, \beta, x)$ describes how machine (Q, A, η) executes one computational step. When in state q and scanning alphabet symbol α on the tape:
  - Machine (Q, A, η) changes to state r.
  - Machine (Q, A, η) rewrites alphabet symbol α as symbol β on the tape.
  - If $x = -1$, then machine (Q, A, η) moves its tape head one square to the left on the tape and is subsequently scanning the symbol in this square.
  - If $x = +1$, then machine (Q, A, η) moves its tape head one square to the right on the tape and is subsequently scanning the symbol in this square.
  - If $r = h$, machine (Q, A, η) enters the halting state h, and the machine halts.

Definition 14. *Turing Machine Tape*
The Turing machine tape T is represented as a function $T : \mathbb{Z} \to A$ where $\mathbb{Z}$ denotes the integers. The tape T is M-bounded if there exists a bound $M > 0$ such that $T(k) = T(j)$ whenever $|k|, |j| \geq M$. The Turing machine definitions in [13] and [45] assume the initial tape, before program execution begins, is M-bounded and the tape contains only blank symbols, denoted here as #, outside the bound. The symbol on the kth square of the tape is $T(k)$.

Definition 15. *Turing Machine Configuration with Tape Head Location*
Let (Q, A, η) be a Turing machine with tape T. A configuration is an element of the set $C = (Q \cup \{h\}) \times \mathbb{Z} \times \{T : T \text{ is tape with range } A\}$. If (q, k, T) is a configuration, then k is called the tape head location.

Consider the configuration $(p, 2, \ldots \#\#\alpha\underline{\beta}\#\# \ldots)$. The 1st coordinate indicates that the Turing machine is in state p. The 2nd coordinate indicates that its tape head is currently scanning tape square 2, denoted as $T(2)$. The 3rd coordinate indicates that tape square 1 contains symbol α, tape square 2 contains symbol β, and all other tape squares contain the # symbol.

Definition 16. *Turing Machine Computational Step*
Given Turing machine (Q, A, η) in current configuration (q, k, T) such that $T(k) = \alpha$. After the execution of one computational step, the new configuration is determined by one and only one of the four cases

1. $(r, k-1, S)$ if $\eta(q, \alpha) = (r, \beta, -1)$ for non-halting state r
2. $(r, k+1, S)$ if $\eta(q, \alpha) = (r, \beta, +1)$ for non-halting state r
3. $(h, k+1, S)$ if $\eta(q, \alpha) = (h, \beta, +1)$ for halting state h
4. $(h, k-1, S)$ if $\eta(q, \alpha) = (h, \beta, -1)$ for halting state h

such that for all four cases the new tape $S(j) = T(j)$ whenever $j \neq k$ and $S(k) = \beta$. In cases (3) and (4), the machine execution halts.

If the machine is currently in configuration (q_0, k_0, T_0) and over the next n steps the sequence of machine configurations (points) is (q_0, k_0, T_0), (q_1, k_1, T_1), (q_2, k_2, T_2), $\ldots, (q_n, k_n, T_n)$, then this execution sequence is sometimes called the next $n+1$ computational steps.

References

1. Harold Abelson and Gerald Jay Sussman. Structure and Interpretation of Computer Programs. Cambridge, MA. MIT Press, (1996)
2. G. Agnew. Random Source for Cryptographic Systems. Advances in Cryptology - EUROCRYPT 1987 Proceedings. Springer Verlag, 77–81 (1988)
3. M. Ajtai, J. Komlós, E. Szemerédi. A note on Ramsey numbers. Journal Combinatorial Theory. Ser. A **29**, 3, 354–360 (1980)
4. M. Ajtai, J. Komlós, E. Szemerédi. A dense infinite Sidon sequence. European J. Combin. **2**, 1, 1–11 (1981)
5. M. Ajtai, J. Komlós, E. Szemerédi. Sorting in $c \log n$ parallel steps. Combinatorica. **3**, 1, 1–19 (1983)
6. N. Alon. Eigenvalues, geometric expanders, sorting in rounds, and Ramsey theory. Combinatorica. **6**, 3, 207–219 (1986)
7. N. Alon. Explicit Ramsey graphs and orthonormal labelings. Electron. J. Combin. **1**, Research Paper 12, 8 pages, (electronic), (1994)
8. N. Alon. The Shannon capacity of a union. Combinatorica. **18**, 3, 301–310 (1998)
9. Robert Axelrod and William D. Hamilton. The Evolution of Cooperation. Science. New Series, Vol. **211**, No. 4489. 1390–1396 (1981)
10. S.A. Burr. Determining Generalized Ramsey Numbers is NP-hard. Ars Combinatoria. **17**, 21–25 (1984)
11. Cristian S. Calude, Michael J. Dinneen, Monica Dumitrescu, Karl Svozil. Experimental Evidence of Quantum Randomness Incomputability. Phys. Rev. A **82**, 022102, 1–8 (2010)
12. S.A. Cook. The complexity of theorem proving procedures. Proceedings, Third Annual ACM Symposium on the Theory of Computing. ACM, New York, 151–158 (1971)
13. Martin Davis. Computability and Unsolvability. Dover Publications, New York, (1982)
14. R.C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. Proceedings of the Sixth International Symposium on Micro Machine and Human Science. Nagoya, Japan, 39–43 (1995)
15. P. Erdös and G. Szekeres. A combinatorial problem in geometry. Compositio Math. **2**, 464–470 (1935)
16. P. Erdös and R. Rado. Combinatorial theorems on classifications of subsets of a given set. Proc. London Math. Soc. **3** 2, 417–439 (1952)
17. Michael Stephen Fiske. Machine Learning. US 7,249,116 B2 (2003) (See http://tinyurl.com/3pcgfvr or http://tinyurl.com/6gcxppd)
18. Michael Stephen Fiske. Effector Machine Computation. US 7,398,260 B2 (2004). Provisional 60/456,715 (2003) (See http://tinyurl.com/6l5wuhz or http://tinyurl.com/6zzly8p)
19. Michael Stephen Fiske. Active Element Machine Computation. US Application 20070288668 (2007) (See http://tinyurl.com/62gv8ke or http://tinyurl.com/6hz8by4)
20. H. Furstenberg. Recurrence in Ergodic Theory and Combinatorial Number Theory. Princeton, NJ. Princeton University Press, (1981)
21. Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, (1979)

22. R. L. Graham and V. Rodl. Numbers in Ramsey Theory. Surveys in Combinatorics, LMS Lecture Note Series 123. Cambridge University Press, (1987)
23. R. L. Graham and B.L. Rothschild. A Survey of Finite Ramsey Theorems. Proc. 2nd Louisiana State Univ. Conference on Combinatorics, Graph Theory and Computation, 21–40, (1971)
24. A. W. Hales and R.I. Jewett. Regularity and positional games. Trans. Amer. Math. Soc. **106**, 222–229 (1963)
25. John Hertz, Anders Krogh and Richard G. Palmer. Introduction To The Theory of Neural Computation. Addison-Wesley Publishing Company. Redwood City, California, (1991)
26. J. H. Holland. Outline for a logical theory of adaptive systems. JACM. **3**, 297–314 (1962)
27. J. H. Holland. Adaptation in Natural and Artificial Systems. Cambridge, MA. MIT Press, (1992)
28. ID Quantique SA. Quantis: Random Number Generation Using Quantum Optics. http://www.idquantique.com/images/stories/PDF/quantis-random-generator/quantis-whitepaper.pdf, (electronic), Geneva, Switzerland, (2001-2011)
29. J.R. Koza. Genetic Programming: On the Programming of Computer by Means of Natural Selection. Cambridge, MA. MIT Press, (1992)
30. Edward A. Lee. Computing Needs Time. Technical Report No. UCB/EECS-2009-30. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-30.html, (electronic). Electrical Engineering and Computer Sciences, University of California at Berkeley (2009)
31. Warren S. McCulloch, Walter Pitts. A logical calculus immanent in nervous activity. Bulletin of Mathematical Biophysics. **5**, 115–133 (1943)
32. B. D. McKay and S. P. Radziszowski. Subgraph counting identities and Ramsey numbers. Journal Combinatorial Theory. Ser. B **69**, 2, 193–209 (1997)
33. Marvin Minsky. Computation: Finite and Infinite Machines (1st edition). Englewood Cliffs, NJ. Prentice-Hall, Inc, (1967)
34. J. Paris and L. Harrington. A mathematical incompleteness in Peano arithmetic. Handbook for Mathematical Logic, editor J. Barwise. North-Holland, (1977)
35. Wilfrid Rall. The Theoretical Foundation of Dendritic Function. Selected Papers of Wilfrid Rall with Commentaries. Edited by Idan Segev, John Rinzel, and Gordon Shepherd. MIT Press. Cambridge, Massachusetts, (1995)
36. F. P. Ramsey. On a problem of formal logic. Proc. London Math. Soc. Series 2 **30**, 264–286 (1930)
37. F. S. Roberts. Applications of Ramsey theory. Discrete Appl. Math. **9**, 3, 251–261 (1984)
38. Abraham Robinson. Non-standard Analysis. Revised Edition. Princeton, NJ. Princeton University Press, (1996)
39. B.L. Rothschild. A generalization of Ramsey's theorem and a conjecture of Erdös. Doctoral Dissertation, Yale University, New Haven, Connecticut, (1967)
40. I. Schur. Uber die Kongruenz $x^m + y^m \equiv z^m (\text{mod } p)$. Deutsche Math. **25**, 114–117 (1916)
41. C. E. Shannon. The zero error capacity of a noisy channel. Institute of Radio Engineers, Transactions on Information Theory. IT-2, 8–19 (1956)
42. Joel H. Spencer. Ten Lectures on the Probabilistic Method. SIAM, page 4 (1994)
43. André Stefanov, Nicolas Gisin, Olivier Guinnard, Laurent Guinnard and Hugo Zbinden. Optical quantum random number generator. Journal of Modern Optics. 1362-3044, **47**, 4, 595–598 (2000)
44. H. E. Sturgis, and J. C. Shepherdson. Computability of Recursive Functions. J. Assoc. Comput. Mach. **10**, 217–255 (1963)
45. Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. Series 2 **42** (Parts 3 and 4), 230–265 (1936). A correction, ibid. **43**, 544–546 (1937).
46. A. C. C. Yao. Should tables be sorted? J. Assoc. Comput. Mach. **28**, 3, 615–628 (1981)